No. 16

PROCEEDINGS OF THE EASTERN JOINT COMPUTER CONFERENCE

PAPERS PRESENTED AT
THE JOINT IRE-AIEE-ACM COMPUTER CONFERENCE
BOSTON, MASSACHUSETTS, DECEMBER 1-3, 1959

Sponsors

THE INSTITUTE OF RADIO ENGINEERS
Professional Group on Electronic Computers

THE AMERICAN INSTITUTE OF ELECTRICAL ENGINEERS
Committee on Computing Devices

THE ASSOCIATION FOR COMPUTING MACHINERY

Published by
THE 1959 EASTERN JOINT COMPUTER CONFERENCE
FOR
THE NATIONAL JOINT COMPUTER COMMITTEE

The Virtual Memory in the STRETCH Computer

JOHN COCKE AND HARWOOD G. KOLSKY†

EARLY in the planning of the STRETCH computer it was seen that by using the latest solid state components in sophisticated circuits it would be possible to increase the speed of floating point arithmetic by almost two orders of magnitude over that in existing computers. However, there seemed to be no possibility of developing on the same time-scale economically feasible large memories with more than a factor of ten or perhaps twenty increase in speed. As a result, the proposed system appeared to be in danger of being seriously memory-access limited.

Moreover, as the speed of the floating point operations increases, a larger and larger percentage of the computer's time is spent on "parasitic operations", *i.e.*, operations whose only function is program control and data selection. It was obvious that a radically new machine organization was necessary in order to capitalize upon the possibilities opened up by the high arithmetic speeds in the presence of relatively slow memories.

At this time, a number of persons were considering the possibility of a "look-ahead" device in which an independent indexing arithmetic unit would prepare the effective addresses of instructions and initiate memory references to a multiplicity of memory boxes. The data thus fetched would be held in high-speed buffer registers until needed by the arithmetic unit. This device would serve two desirable purposes: (1) some of the parasitic operations would be done in parallel and thus not delay the principal calculations, and (2) several memory boxes could be running simultaneously, giving the effect of higher memory speed.

Since our original work on the virtual memory and simulation in 1957–58, a large number of detailed changes have been made in the actual hardware design of STRETCH. These necessitated several modifications in the simulation program to estimate their effect on the overall system performance. In this report we are omitting many of these changes for expository reasons, since our purpose is to describe the virtual memory and timing simulation concepts, not to describe the STRETCH hardware exactly. The result is that the system described below embodies a more general system than that found in the simulator, which in turn is more general than that found in the actual computer.

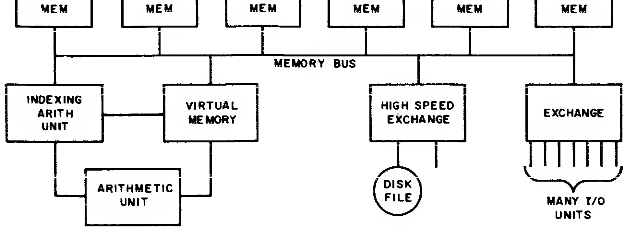


Fig. 1—Schematic of Stretch computer.

† International Business Machines Corporation, Poughkeepsie, New York.

General Description of the System

The major logically-independent blocks of the STRETCH computer are shown in Fig. 1. Each of the units pictured may be considered as operating asynchronously. That is, each does its tasks as fast as possible independently of the others. In theory, each box could have its own clocking circuits and still operate properly. In practice, for economy's sake they are all timed by the same master oscillator, but this does not destroy their *logical* independence.

The bus control unit serves as a routing agent between the memories and the various data processing units. If two or more units make a request simultaneously the control unit assigns priorities in the following order: (1) High-speed Exchange, (2) Basic Exchange, (3) Virtual Memory, and (4) Indexing Arithmetic Unit.

The Indexing Arithmetic Unit fetches instructions, performs all necessary indexing operations and sends the instructions to be executed to the Virtual Memory.

The Virtual Memory fetches and receives the data required by the instruction and holds this data until the arithmetic unit is ready for it. The Virtual Memory also performs all store operations. It holds the data generated by the arithmetic unit or indexing arithmetic unit until the memory to which the data must be sent is available. Thus the virtual memory acts not only as a "look-ahead" for instructions to be fed to the arithmetic unit, but also acts as a "look-behind" storage buffer.

The actual design of such a "look-ahead" device posed a number of logical problems, particularly in connection with conditional branches. However, a machine organization of this complexity requires a detailed timing analysis in order to determine the value of adding hardware in the form of the virtual memory. This is especially true since the sole function of the virtual memory is to increase machine speed, by increasing the efficiency of other devices. It was also felt that the timing analysis could not be made on the basis of a few trivial examples (e.g. matrix multiply). Machine performance obtained in this fashion can be extremely deceptive. Since a detailed timing analysis of a computer of this complexity is extremely tedious to carry out by hand, it became clear that if the job were to be done, it would be necessary to simulate the proposed machine on another computer. This prompted us to write the simulation program to be described later.

With the above general organization in mind, let us discuss some of the logical problems posed by such a system. The first problem is a result of the very concept which enables us to obtain such great benefits from the stored program computer — the ability to treat instructions as data. In a system such as we have proposed there is a large amount of simultaneous operation. For example, the indexing arithmetic unit may be busy preparing an instruction before previous instructions have been completed or even started by the arithmetic unit. One of these previous instructions may modify the instruction which is presently being indexed. The virtual memory must recognize this situation and allow the intervening instructions to be completed before doing the modified instruction.

A similar problem exists with respect to ordinary data. In order to operate several memories simultaneously, it is necessary to start obtaining data from these memories before the preceding operations have been completed. Yet, one of these operations may be a store into one of the data locations. The virtual memory must make provisions to insure that each instruction obtains the most up-to-date data as implied by the order of the program.

One of the novel features of the STRETCH computer is its elaborate interrupt system. Under this system, whenever some unexpected occurrence arises, the program will be interrupted and control will pass to a special routine which is designed to take care of the case in question, then return control to the original program. In this situation the virtual memory must have provisions to retain enough information so that when an interrupt occurs we can resume the computation exactly where we left off. It must be able to recognize which of the changes that have been made in advance are not desired and should be obliterated, and which are exact solutions that must be restored.

Another special case arises when a conditional branch on arithmetic results occurs. Here we will not know which of the two branches we should have taken until the preceding instruction is executed. In the case where the wrong path has been selected, the virtual memory must be prepared to drop the intermediate results which have been computed and pick up the correct branch in a way very similar to that of an interrupt.

Summing up all these logical problems, we may state that the fundamental rule for the virtual memory is that it must make the asynchronous and non-sequential computer give results identical to those which would be obtained by performing the program one instruction at a time in the order in which they are written.

Definitions

Operations

Operations are considered to be of three types:

(1) Bring or Fetch Type — All instructions requiring data to be transmitted from external memory to the virtual memory.

(2) Store Type — Instructions requiring the transmission of data from the virtual memory to external memory or index memory.
(*Note:* We consider all indexing instructions to be of the store type, although the store may be to either external memory or index memory.)

(3) Immediate Type — All operations not requiring data transmission.

Virtual Memory Quantities

(1) Virtual Memory — A number of virtual memory (or look-ahead) levels (numbered 0 to $N - 1$).

(2) Level of Virtual Memory — A collection of registers and control bits. The contents of the jth level are shown in Fig. 2.

(3) Instruction Address Register (I_j) — Contains the address of the instruction currently in the jth level.

(4) Operation Code Register (OP_j) — Contains the operation to be performed by the arithmetic unit.

(5) Store Bit (S_j) — a one-bit trigger which indicates the level, contains a store type instruction.

(6) Bring Bit (B_j) — A one-bit trigger which indicates the level, contains a fetch type instruction for which the data access has not been started.

(7) Forwarding Bit (F_j) — A one-bit trigger which indicates that the jth level must transmit data to another level.

(8) Forwarding Address (FA_j) — A register which contains the number of the level to which the data must be sent if F_j is set.

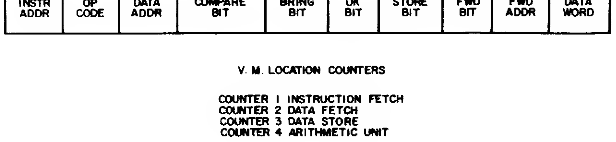


Fig. 2—Virtual memory — contents of one level.

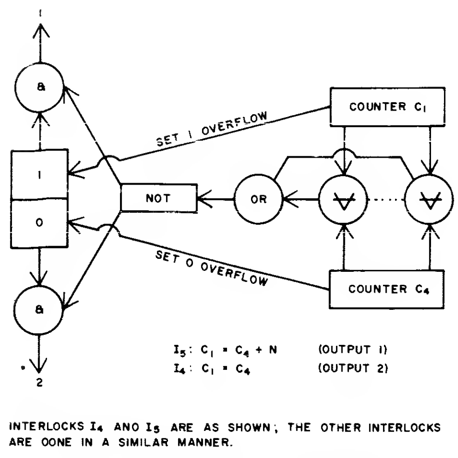

a
1
0
a
COUNTER C1
SET 1 OVERFLOW
NOT
OR
∀
∀
SET 0 OVERFLOW
COUNTER C4
I5: C1 = C4 + N (OUTPUT 1)
I4: C1 = C4 (OUTPUT 2)
1
2
INTERLOCKS I4 AND I5 ARE AS SHOWN; THE OTHER INTERLOCKS ARE DONE IN A SIMILAR MANNER.

Fig. 3—Virtual memory interlocks.

(9) O. K. Bit (OK_j) — A trigger which when set indicates that the correct data for the instruction to be executed is present in the jth data field.

(10) Data Field (D_j) — A register which contains the operand data for the instruction.

(11) Data Address (DA_j) — The operand data address (already indexed by the IAU) for D_j.

(12) Compare Bit (C_j) — A trigger which if not set indicates the address in DA_j should not be included in any address comparisons being made.

Counters

The virtual memory is controlled by a set of counters which count mod(N), where N is the number of virtual memory levels.

(1) Counter one (C_1) — Indicates the level into which the next instruction may be placed.

(2) Counter two (C_2 — Indicates the level from which the next bring type instruction may be initiated.

(3) Counter three (C_3) — Indicates the level from which the next store type instruction may be initiated.

(4) Counter four (C_4) — Indicates the level from which the arithmetic unit will get its next operation and data.

Interlocks

The above counters must be interlocked in the following manner to assure proper sequential operation of the computer (see Fig. 3:)

(1) Interlock one (I_1): $C_1 = C_3 + N$ Prevents the IAU from placing the next operation into the level indicated by C_1 because an unexecuted store is still in the level.

(2) Interlock two (I_2): $C_1 = C_3$ Prevents a store from being initiated from the level indicated by C_3 because the store has already been done.

(3) Interlock three (I_3): $C_1 = C_2$ Similar to I_2, prevents a fetch from being initiated.

(4) Interlock four (I_4): $C_1 = C_4$ Prevents the arithmetic unit from executing an old instruction.

(5) Interlock five (I_5): $C_1 = C_4 + N$ Prevents the IAU from placing the next instruction into the level indicated by C_1 because the instruction there has not been executed yet.

Logic of the Virtual Memory

There are two basic precepts which must be kept in mind to understand the operation of the virtual memory:

(1) The OK bit (O_j) being set in the jth level indicates that the contents of D_j is the correct data called for by DA_j. All operations will be performed only under this condition, and logical decisions will be made in such a manner as to make sure this is the case.

(2) Addresses can be compared by the IAU with every DA_j address simultaneously. DA_j is not used for any level which does not have its C_j bit set. If a comparison exists between a new DA_j being placed in the virtual memory and an old DA_k, the compare bit C_k is turned off and the address of level j is placed in FA_k. This insures a unique meaning for the comparison. If this were not done, another instruction address DA_e might compare against *two* levels and thus cause an ambiguity.

Instruction Fetch Logic

Fig. 4 is a flow diagram of the IAU Instruction Fetch Procedure. The logic is as follows: If the IAU is ready to fetch another instruction, it compares the instruction address with all the DA_j's of virtual memory. If there is no comparison, the instruction fetch is initiated. If there is a comparison, the IAU

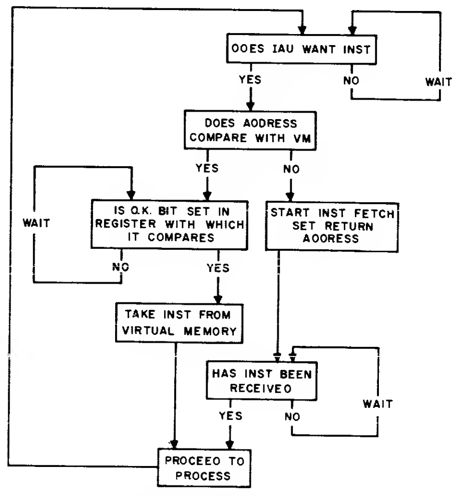

DOES IAU WANT INST
YES
NO
WAIT
DOES ADDRESS COMPARE WITH VM
YES
NO
WAIT
IS O.K. BIT SET IN REGISTER WITH WHICH IT COMPARES
START INST FETCH SET RETURN ADDRESS
NO
YES
TAKE INST FROM VIRTUAL MEMORY
HAS INST BEEN RECEIVED
YES
NO
WAIT
PROCEED TO PROCESS

Fig. 4—Instruction fetch procedure.

must take its instruction from the virtual memory provided the OK bit is set; otherwise, it must wait until the OK bit is set.

Note: This procedure prevents the logical difficulty mentioned earlier which would occur if the virtual memory contained a store order into the instruction presently being fetched.

For Example:	a	STORE Address $a + 2$
	$a + 1$	LOAD M, i
	$a + 2$	ADD N, i
	$a + 3$	----

The store to $a + 2$ must be done in sequence or the old value N would be used for the address instead of the quantity being set by a.

Indexing Logic

Fig. 5 shows the flow for instruction indexing. After determining that an instruction is ready to be indexed, the IAU tests whether or not the index value is available. If it is, the indexing operation is started; if not, the memory reference is started and the IAU waits until the data returns before proceeding. If the index-fetch has not been started, the IAU compares the index address against all the data addresses in virtual memory. If none compare, the index value is fetched normally. If one does compare, the index fetch is held up until the OK bit is set for the data. This value from the virtual memory is then used for indexing the instruction.

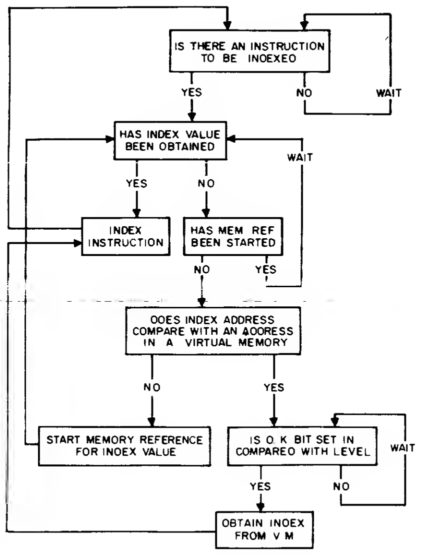

IS THERE AN INSTRUCTION TO BE INDEXED
YES
NO
WAIT
HAS INDEX VALUE BEEN OBTAINED
WAIT
YES
NO
INDEX INSTRUCTION
HAS MEM REF BEEN STARTED
NO
YES
DOES INDEX ADDRESS COMPARE WITH AN ADDRESS IN A VIRTUAL MEMORY
NO
YES
START MEMORY REFERENCE FOR INDEX VALUE
IS O K BIT SET IN COMPARED WITH LEVEL
WAIT
YES
NO
OBTAIN INDEX FROM V M

Fig. 5—Indexing procedure.

Logic of Putting Instructions in the Virtual Memory

(1) Figs. 6, 6A, 6B, 6C represent the logical flow for putting instructions into the virtual memory. If the indexing arithmetic unit has an instruction prepared for the virtual memory, it may transmit the instruction into the virtual memory if interlocks one and five do not forbid it. These interlocks prohibit a new instruction from destroying an old one which has not been executed as yet, whether an arithmetic operation (I_5) or an unexecuted store (I_1). The handling of the instructions varies depending on whether they are of the bring type, store type, or immediate type.

(2) The bring type, as described in Fig. 6A, proceeds as follows: If the effective data address of the instruction compares with the *DA* address in some level, the instruction, its op code, and effective data address are loaded into the level marked by C_1. The compare bit for level C_1 is set to one while the compare bit for the compared-with level is set to zero. If the OK bit in this compared-with level is set, meaning that the data located there is correct, the data is transmitted directly to the C_1 level and its OK bit is also set. If the OK bit is not set, we must tag the compared-with level by setting its forwarding bit and by putting the value of C_1 into its forwarding address; the bring bit for level C_1 is also set to *zero* since no further data fetch is required.

If the effective data address does not compare with any Virtual Memory level, the instruction is put directly into level C_1, its OK bit is set to *zero*, and its bring bit is set to *one*, indicating that a fetch must be started.

(3) Fig. 6B shows the store type procedure. If the effective address of the instruction does not compare with the *DA* address in some level, the instruction is placed into the level marked by C_1. The store bit is set to one indicating that a store will be required. The level's bring bit and forwarding bit are set to *zero*; its compare bit is set to *one*. If on the other hand the addresses do compare, the same procedure is followed; but in addition, the compare bit in the level compared-with is set to zero so that future comparisons will not use it.

The OK bit has not yet been set. It is set to *one* if the operation is an index store and set to *zero* if it is an ordinary store. For the ordinary store it is clear that the OK bit should be *zero* since the data must come from the arithmetic unit after the preceding instruction is executed.

As was mentioned in the definition previously we treat all indexing instructions as store type and place the new value of the indexed quantity into the virtual memory. This is done because the indexing arithmetic unit is going ahead of the normal order of instruction execution and an interruption may occur before this indexing instruction should have

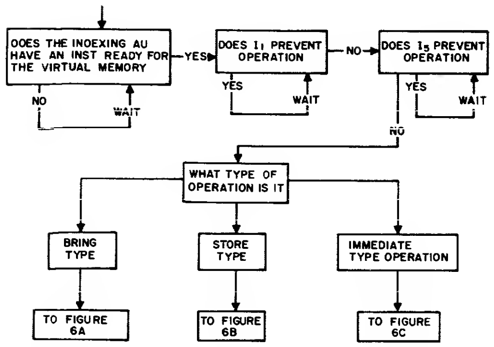

DOES THE INDEXING AU HAVE AN INST READY FOR THE VIRTUAL MEMORY
YES
NO
WAIT
DOES I1 PREVENT OPERATION
NO
YES
WAIT
DOES I5 PREVENT OPERATION
YES
WAIT
NO
WHAT TYPE OF OPERATION IS IT
BRING TYPE
STORE TYPE
IMMEDIATE TYPE OPERATION
TO FIGURE 6A
TO FIGURE 6B
TO FIGURE 6C

Fig. 6—Procedure for placing instructions into the virtual memory.

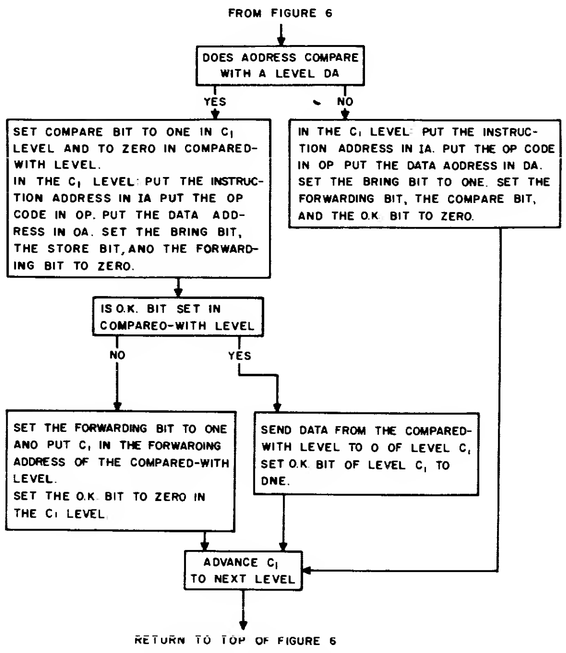

FROM FIGURE 6
DOES ADDRESS COMPARE WITH A LEVEL DA
YES
NO
SET COMPARE BIT TO ONE IN C1 LEVEL AND TO ZERO IN COMPARED-WITH LEVEL. IN THE C1 LEVEL: PUT THE INSTRUCTION ADDRESS IN IA PUT THE OP CODE IN OP. PUT THE DATA ADDRESS IN OA. SET THE BRING BIT, THE STORE BIT, AND THE FORWARDING BIT TO ZERO.
IN THE C1 LEVEL: PUT THE INSTRUCTION ADDRESS IN IA. PUT THE OP CODE IN OP. PUT THE DATA ADDRESS IN DA. SET THE BRING BIT TO ONE. SET THE FORWARDING BIT, THE COMPARE BIT, AND THE O.K. BIT TO ZERO.
IS O.K. BIT SET IN COMPARED-WITH LEVEL
NO
YES
SET THE FORWARDING BIT TO ONE AND PUT C1 IN THE FORWARDING ADDRESS OF THE COMPARED-WITH LEVEL. SET THE O.K. BIT TO ZERO IN THE C1 LEVEL.
SEND DATA FROM THE COMPARED-WITH LEVEL TO D OF LEVEL C1 SET O.K. BIT OF LEVEL C1 TO ONE.
ADVANCE C1 TO NEXT LEVEL
RETURN TO TOP OF FIGURE 6

Fig. 6(a)—Logical conditions for bring type operations.

been done. In this case, the old value of the index is still in the index register. On the other hand the indexing arithmetic unit compares with the virtual memory and extracts the most recent value of the index for indexing succeeding instructions. The OK bit is set to *one* since the appropriate data is in the above level. Both the new and old index values must be carried along to give logically correct conditions in the case of an interrupt. A situation very similar to interrupt occurs in branches on arithmetic results where the indexing arithmetic unit "guesses" which branch will be taken and proceeds with fetching and processing the instructions on this branch, subject to being wiped out if the guess proves to be wrong. (See the discussion on "Wrong way Branches" below.)

(4) Immediate type instructions are the simplest type because they essentially carry their data with them. Fig. 6C shows the logic in this case.

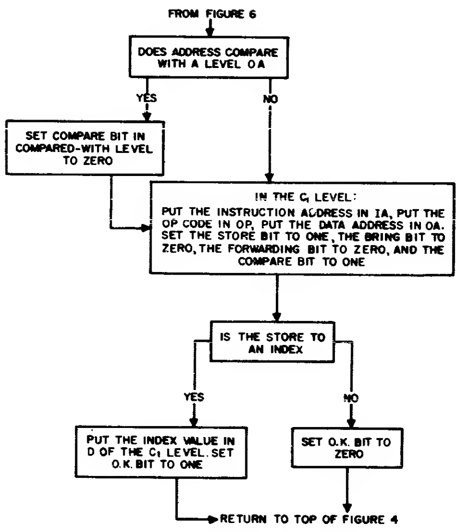

FROM FIGURE 6
DOES ADDRESS COMPARE WITH A LEVEL DA
YES
NO
SET COMPARE BIT IN COMPARED-WITH LEVEL TO ZERO
IN THE C1 LEVEL: PUT THE INSTRUCTION ADDRESS IN IA, PUT THE OP CODE IN OP, PUT THE DATA ADDRESS IN DA. SET THE STORE BIT TO ONE, THE BRING BIT TO ZERO, THE FORWARDING BIT TO ZERO, AND THE COMPARE BIT TO ONE
IS THE STORE TO AN INDEX
YES
NO
PUT THE INDEX VALUE IN D OF THE C1 LEVEL. SET O.K. BIT TO ONE
SET O.K. BIT TO ZERO
RETURN TO TOP OF FIGURE 4

Fig. 6(b)—Logical conditions for store type operations.

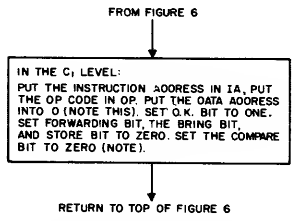

FROM FIGURE 6
IN THE C1 LEVEL: PUT THE INSTRUCTION ADDRESS IN IA, PUT THE OP CODE IN OP. PUT THE DATA ADDRESS INTO D (NOTE THIS). SET O.K. BIT TO ONE. SET FORWARDING BIT, THE BRING BIT, AND STORE BIT TO ZERO. SET THE COMPARE BIT TO ZERO (NOTE).
RETURN TO TOP OF FIGURE 6

Fig. 6(c)—Logical conditions for immediate type operations.

The instruction is placed in the virtual memory level marked by C_1. The address field of the instruction is placed in the data field of C_1. The OK bit is set to *one* indicating the data is present. The bring and store bits are both set to *zero*. The compare bit is set to *zero* since the DA address field has no meaning for immediate type ops. (The data address of the last instruction which occupied this level still remains in DA, so it has no relation to the present *D* field.)

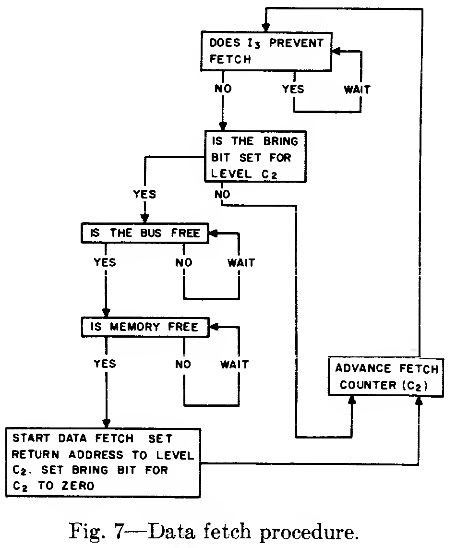

DOES I3 PREVENT FETCH
NO
YES
WAIT
IS THE BRING BIT SET FOR LEVEL C2
YES
NO
IS THE BUS FREE
YES
NO
WAIT
IS MEMORY FREE
YES
NO
WAIT
ADVANCE FETCH COUNTER (C2)
START DATA FETCH SET RETURN ADDRESS TO LEVEL C2. SET BRING BIT FOR C2 TO ZERO

Fig. 7—Data fetch procedure.

Logic of Data Fetching

See Fig. 7: When an instruction of the bring type has been placed in the virtual memory, the data required by the instruction in general will not be present (unless a comparison exists as was described above) and thus the data must be obtained from core storage. The fetch cannot be started if interlock I_3 holds, which means all the fetches corresponding to the instructions presently in the virtual memory have been started. If a fetch is possible, the bring bit at level C_2 indicates whether or not a fetch is necessary. If necessary the fetch may be started if the memory bus and memory unit corresponding to the data address are not already being used. When the fetch is started, the bring bit for level C_2 is set to zero. The counter C_2 is then stepped forward to the next level.

Logic of Data Storing

Fig. 8 shows the Data Store Logic, which is very similar to that for data fetching just described. The only significant difference is that the OK bit must be set before the operation can be started.

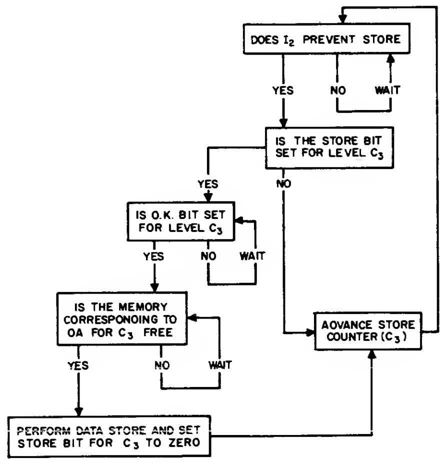

DOES I_2 PREVENT STORE
YES
NO
WAIT
IS THE STORE BIT SET FOR LEVEL C_3
YES
NO
IS O.K. BIT SET FOR LEVEL C_3
YES
NO
WAIT
IS THE MEMORY CORRESPONDING TO DA FOR C_3 FREE
YES
NO
WAIT
ADVANCE STORE COUNTER (C_3)
PERFORM DATA STORE AND SET STORE BIT FOR C_3 TO ZERO

Fig. 8—Data store procedure.

Logic for Placing Data into the Virtual Memory

In Fig. 9, we see the logical conditions which must be satisfied by the data returning from memory addressed to the virtual memory. The return address which was supplied when the fetch was started selects the level into which the data will be placed. The OK bit is then set to *one*, indicating that the proper data is in the level. The operation is complete at this point unless the forwarding bit is set. In this case, the data must be forwarded to the level designated by the forwarding address. This procedure continues from level to level as long as the data continues to arrive into a level whose forwarding bit is set. This procedure automatically supplies all operands present having identical data addresses with the proper data, without additional memory references.

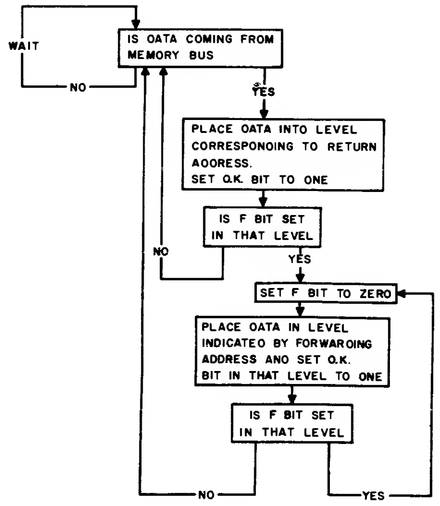

WAIT
IS DATA COMING FROM MEMORY BUS
NO
YES
PLACE DATA INTO LEVEL CORRESPONDING TO RETURN ADDRESS. SET O.K. BIT TO ONE
IS F BIT SET IN THAT LEVEL
NO
YES
SET F BIT TO ZERO
PLACE DATA IN LEVEL INDICATED BY FORWARDING ADDRESS AND SET O.K. BIT IN THAT LEVEL TO ONE
IS F BIT SET IN THAT LEVEL
NO
YES

Fig. 9—Procedure for placing data into virtual memory.

Logic of Removing Instructions from the Virtual Memory

In Fig. 10, we notice that as the arithmetic unit completes an instruction it checks to see if the next instruction in the virtual memory is ready to be executed (indicated by interlock I_4). Note that the operation may be an unconditional branch, a conditional branch, or an index type store, as well as a normal bring or store type instruction involving the accumulator. Fig. 10 shows only the cases which involve the universal accumulator. Instructions such as the unconditional branches are merely ignored at this point. They are carried along only to provide the data for recovery in the event an interrupt occurs. The execution of the conditional branches on arithmetic results are described in the next section.

If the next instruction marked by counter C_4 is ready, it is fed into the arithmetic unit. If it is a store

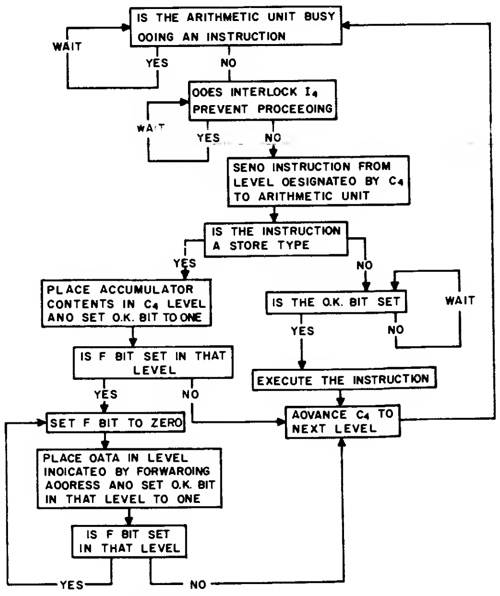

IS THE ARITHMETIC UNIT BUSY DOING AN INSTRUCTION
WAIT
YES
NO
DOES INTERLOCK I_4 PREVENT PROCEEDING
WAIT
YES
NO
SEND INSTRUCTION FROM LEVEL DESIGNATED BY C_4 TO ARITHMETIC UNIT
IS THE INSTRUCTION A STORE TYPE
YES
NO
PLACE ACCUMULATOR CONTENTS IN C_4 LEVEL AND SET O.K. BIT TO ONE
IS THE O.K. BIT SET
WAIT
YES
NO
IS F BIT SET IN THAT LEVEL
YES
NO
EXECUTE THE INSTRUCTION
SET F BIT TO ZERO
ADVANCE C_4 TO NEXT LEVEL
PLACE DATA IN LEVEL INDICATED BY FORWARDING ADDRESS AND SET O.K. BIT IN THAT LEVEL TO ONE
IS F BIT SET IN THAT LEVEL
YES
NO

Fig. 10—Procedure for removing instructions from virtual memory.

type, the data is gated from the accumulator into the data field of level C_4, and the OK bit is set to one. If the forwarding bit of the level is set, a forwarding procedure in this case is *essential* for the proper logical operation of the computer, whereas in the bring case it is a time-saver only.

If the instruction is not a store type, the arithmetic unit must hold up until the OK bit for the level is set. When the OK bit is set, the instruction is gated into the arithmetic unit and executed.

Logic of Interrupt Procedure

If for any cause an interrupt (or trap) from a special condition occurs, the instruction which is being executed in the arithmetic unit is completed. However, the next instruction is not executed in spite of the fact all the data preparation for it may have been completed. The address in the *IA* (instruction address) field will serve as the value to reset the instruction counter if it is desired.

The Virtual Memory is initialized, *i.e.*, set to the starting conditions of an interrupt, with the exception that all store orders which have already received data from the accumulators must be executed first. If the interrupt is of such a nature that the normal flow of instructions is not resumed, the procedure of storing the modified values of the index registers in the Virtual Memory gives logically correct results, *i.e.*, the same as if the interrupt had occurred before the indexing took place.

Description of Timing Simulation Program

During the logical design of STRETCH it was necessary to prove the value of the virtual memory concept and to assist in the selection of optimum values of various system design parameters. Examples of such parameters are: The number of memory boxes, interlace and allocation of memory addresses, and numbers of virtual memory levels. Also of interest were trade-off factors for speeds of indexing arithmetic unit, memories, etc.

In November 1957 the Timing Simulator (SIM-2) described here was written for the IBM 704. This program attempted to answer such questions quantitatively by simulating the time-wise operation of STRETCH on typical test programs coded in STRETCH language.

The basic logic of the 704 program follows the principles just described in the preceding section for the virtual memory. It should be stressed that the simulator is a *timing* simulator and does not execute the instructions in an arithmetic sense. It traces the time-wise progress of the instructions through the components of the computer, observing all the interlocks and time delays necessary for correct representation of the behavior of the machine.

One of the fundamental concepts in the STRETCH design is that of *asynchronous* operation of the components. This means that there are a large number of logical steps being executed at any one time in the computer, each of them proceeding at its own rate. To simulate this flow of many parallel continuous operations, we have broken the continuous time variable into finite time steps. The basic time step is taken as 0.1 microsecond in the simulator.

By taking 0.1 microsecond as our quantum of time, we are automatically setting the scale of the smallest circuit entities which we will consider as being those which accomplish complete functions in 0.1 microsecond or few multiples thereof. Thus, by using this pholosophy, and considering many of the components of the computer as "black boxes", we greatly simplify the details which must be considered without introducing serious timing inaccuracies.

Our experiênce has indicated that more information was gained by making a large number of fast parameter studies using different configurations and programs than could have been obtained by a very slow, detailed simulation of a few runs with more precision per run. Even so, our time scale is *too* fine to make serious input-output application studies. These would require a simpler simulator having at least a factor of 10 coarser basic time interval.

Logic of the Simulator

In the asynchronous organization of STRETCH there can be many major components operating at any one time. To achieve this parallel effect in the simulator we essentially "hold time still" and scan the entire machine representation at each time step. Although every major block of the program is traversed at each time step, if there is no activity required in a given block, only a few tests need be made by the code.

If in this process it is determined that a given logical unit should do an operation, the time interval required for the operation is obtained from a table of constants. The speed of the various logical units can thus be changed parametrically by changing the values in the tables. A constant obtained from the tables is inserted into a memory location called the time counter for that unit. At each time step the program reduces this counter by one until it reaches zero. Thus, the fact that the counter is non-zero can be used to indicate that the particular logical unit is busy and not available to service other requests. When the counter is zero the unit can consider a new input.

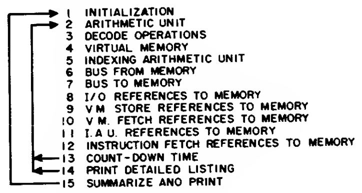
In addition to the time counters many of the logical blocks contain other conditions or interlocks which affect the operation of the block. These conditions are stored in the program and tested before action is undertaken.

It is interesting to note that since the simulator simulates timing only, the sequence of instructions to be executed must be furnished as a "string" with all loops unwound. However, to make the computer behave as it actually would, the loops must be furnished with "wrong way" paths given for the cases where the computer would take such paths. Also one must furnish *more* than enough information along such paths since it is difficult to predict in advance how far the computer will get down the wrong path before it it called back.

Parameters are changed from one run to another by use of control cards. The control cards are set up in such a way that any number of parameters may be changed between runs. Results are given either as detailed timing charts or as summary listings for each problem. The usual procedure has been to print only summary results while making a series of parameter studies. The detailed timing charts as printed on the 704 for most problems would be about 50 feet long for each run. Since over 1000 cases have been run, it is clear that only a few cases could be printed in full detail. These are particularly useful in seeking the causes of conflicts which slow the computer.

Results of Parameter Studies

When the simulator program was completed, we undertook a series of studies in which the main parameters describing the STRETCH system were varied one or two at a time in order to get a measure for the importance of different effects. After this we began to specialize the studies towards answering specific questions in the STRETCH design.



1 INITIALIZATION
2 ARITHMETIC UNIT
3 DECODE OPERATIONS
4 VIRTUAL MEMORY
5 INDEXING ARITHMETIC UNIT
6 BUS FROM MEMORY
7 BUS TO MEMORY
8 I/O REFERENCES TO MEMORY
9 VM STORE REFERENCES TO MEMORY
10 V.M. FETCH REFERENCES TO MEMORY
11 I.A.U. REFERENCES TO MEMORY
12 INSTRUCTION FETCH REFERENCES TO MEMORY
13 COUNT-DOWN TIME
14 PRINT DETAILED LISTING
15 SUMMARIZE AND PRINT

Fig. 11—SIM — 2 simplified flow diagram.

The simplified flow diagram in Fig. 11, indicates the order in which the subroutines for the various logical units are executed at each time step. Using the types of techniques just described above, the logical subroutines simulate the action of the components of the computer such as the virtual memory, arithmetic unit, etc.

Some Results of the Simulation Studies

Fig. 12 shows examples of the type of output listings given by the simulator. Fig. 12 is a piece of a long timing chart with each line of printing representing 0.1 microsecond of time. The columns represent the various components of the computer. On the left and right are timing counts subdividing each microsecond. On the far right are conflict indicators (C on the charts) and waiting indicators, W, which indicate when interlocks prevent operations from proceeding.

The 2nd column, II, gives the number of the instruction being indexed. The 4th column, AU, gives the number of the instruction using the arithmetic unit. The next four columns represent the instructions using the memory buses. The columns labeled X- F-, and M- represent the index, fast, and main memories. A string of X's in the columns represents the cycle time of the memory. The number indicates the instruction using the memory and the number of times which it is repeated gives the readout time of the memory. The columns L- indicate which instruction is located in the virtual memory levels. The other columns are for details in analysis and need not be considered here.

Five of the test problems used most frequently are described below. Other test problems were used for specific studies, but since the results were similar for all problems of a given type, we gradually discontinued using them. The following were originally selected as being typical of different classes of problems.

(1) *Mesh Problem* — Part of an hydrodynamics problem from Los Alamos. It contains a more or less "average" mixture of instructions for scientific problems: 85% floating point instructions, 14% index modification instructions, and 1% VFL. It is usually arithmetic unit limited.

(2) *Monte Carlo Branching Problem* — Part of an actual Monte Carlo neutron diffusion code. It represents a chain of logical decisions with very little arithmetic in between. It contains 47% floating point, 15% index modification instructions, and 36% branches of the indicator and unconditional types. It is largely instruction-access limited.

(3) *Reactor Problem* — The inner loop of a neutron diffusion problem. It consists of 90% floating point arithmetic (39% of which are multiplys) and 10% index modification instructions. It is almost entirely arithmetic unit limited.

(4) *Computer Test Problem* — The evaluation of a polynominal using computed indices. It has 71% floating point, 10% index modification, 6% VFL and 13% indicator branches. It is usually arithmetic unit limited, but not for all configurations.

(5) *Simultaneous Equations* — The inner loop of a matrix inversion routine 67% floating point and 33% index modification. Arithmetic and logic are about equally important. It is limited both by arithmetic and instruction-access speeds.

Speed vs. Number of Levels of Virtual Memory

Fig. 13 shows the effect on computer performance

	II	IS	AU	IF	IM	OF	OM	X1	X2	F1	F2	F3	F4	M1	M2	M3	M4	M5	M6	M7	M8	L1	L2	L3	L4	L5	L6	L7	L8	FD	MD	MC		
1	1																													1		2	1	W
2	1																													1		1	2	CW
3	1																													1			3	CW
4	1					1																								3		2	4	W
5	1					1																								3		1	5	W
6	1										1X																			3			6	W
7	1					3					1X																					2	7	W
8	1					3					1X																					1	8	W
9	1									3X	1X																						9	W
10	1			1						3X	X																					2	10	W
1	1			1						3X	X																					1	1	W
2	1									3X																							2	W
3	1			3						X																						2	3	W
4	1	1		3						X																						1	4	W
5	1	1																															5	W
6	1	2																														2	6	W
7	1	4																														1	7	W
8		1																															8	W
9	2	1																							1							2	9	
10	2	2																							1							1	10	
1	2	4	1					x																	1								1	W
2		1																							1							2	2	W
3	3	1																						2	1					5		1	3	
4	3	2																						2	1					5			4	
5	3	4	2			5		x																2	1							2	5	W
6		1				5																		2	1							1	6	W
7	4	1									5X												3	2	1								7	
8	4	2									5X												3	2	1							2	8	
9	4	2	3					x			5X												3	2	1							1	9	W
10	4	4									5X												3	2	1								10	W
1		1		5							X												3	2	1							2	1	W
2		1		5							X											4	3	2	1					7	4	1	2	
3		1																				4	3	2	1					7	4		3	W
4	5	1				7	4															4	3	2	1							2	4	W
5	5	2				7	4															4	3	2	1							1	5	W
6	5	2								7X				4X								4	3	2	1								6	W
7	5	4								7X				4X								4	3	2	1							2	7	W
8		1								7X				4X								4	3	2	1							1	8	W
9	6	1								7X				4X								4	3	2	5						5		9	W
10	6	2		7						X				4X								4	3	2	5						5	2	10	W
1	6	4		7						X				4X								4	3	2	5						5	1	1	W
2		1												4X								4	3	2	5						5		2	W
3	7	1					5							4X								4	3	6	5					9		2	3	W
4	7	2			4		5							X								4	3	6	5					9		1	4	W
5	7	2			4									X	5X							4	3	6	5					9			5	W
6	7	4				9								X	5X							4	3	6	5							2	6	
7		1	4			9								X	5X							4	3	6	5							1	7	
8	8	1	4								9X			X	5X							4	7	6	5						7		8	
9	8	2									9X			X	5X							4	7	6	5						7	2	9	W
10	8	2						x			9X			X	5X							4	7	6	5						7	1	10	W
1	8	2									9X			X	5X							4	7	6	5						7		1	W
2	8	4		9			7				X			X	5X							4	7	6	5							2	2	W
3		1		9	5		7				X			X	X							4	7	6	5							1	3	W
4		1			5									X	X		7X					8	7	6	5					11			4	C W
5	9	1												X	X		7X					8	7	6	5					11		2	5	C
6	9	2	5												X		7X					8	7	6	5					11		1	6	C
7	9	4	5												X		7X					8	7	6	5					11			7	C
8		1	5			11									X		7X					8	7	6	5							2	8	C
9	10	1	5			11									X		7X					8	7	6	9							1	9	C
10	10	2	5							11X					X		7X					8	7	6	9								10	C
1	10	2	5					x		11X					X		7X					8	7	6	9							2	1	C
2	10	2			7					11X					X		X					8	7	6	9							1	2	CW
3	10	4	6		7			x		11X					X		X					8	7	6	9								3	C
4		1		11						X					X		X					8	7	6	9							2	4	C
5		1	7	11						X							X					8	7	10	9					13		1	5	C
6		1	7														X					8	7	10	9					13			6	C

Fig. 12—Listing of simulator print-out.

of varying the number of levels of virtual memory. Curves for the Monte Carlo and Mesh Calculations with two sets of arithmetic and indexing arithmetic speeds are shown. The *AU* times given are averages for all operations.

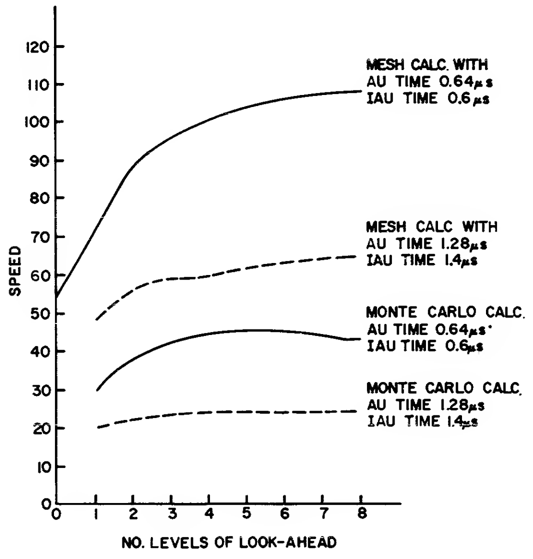


Fig. 13—Computer speed vs. no. of levels of look-ahead registers; 4 main mems. 2.0 μs; 2 fast mems. 0.6 μs for two sets of arith. speeds.

A number of interesting results are apparent from these curves:

(1) There is a tremendous gain to be had in going to the virtual memory organization. The point for "0 levels" means that the arithmetic unit is tied directly to the instruction preparation unit, although simple Indexing-Execution overlap is still possible.

(2) The gain in performance goes up very rapidly for the first two levels, then rises more slowly for the rest of the range.

(3) A large number of levels does the Monte Carlo problem less good than the Mesh problem because constant branching in the former spoils the flow of instructions. Notice that the curve for the Monte Carlo problem actually *decreases* slightly beyond six levels. This phenomenon is a result of memory conflicts caused by extraneous memory references started by the computer running ahead on the wrong-way paths of branches.

(4) The computer performance on a given problem is clearly less for slower arithmetic speeds. However, it is important to note that the *sensitivity* of the performance is also less for slower arithmetic speeds. The virtual memory improves the performance in either case, but it is not a substitute for a fast arithmetic unit.

Speed vs. Number of Main Memory Units

Fig. 14 shows how internal computer performance varies with the total number of memory units for a particular problem. The entire calculation is assumed to be contained in memory for all cases. The speed gain from overlapping memories is quite apparent from the graphs.

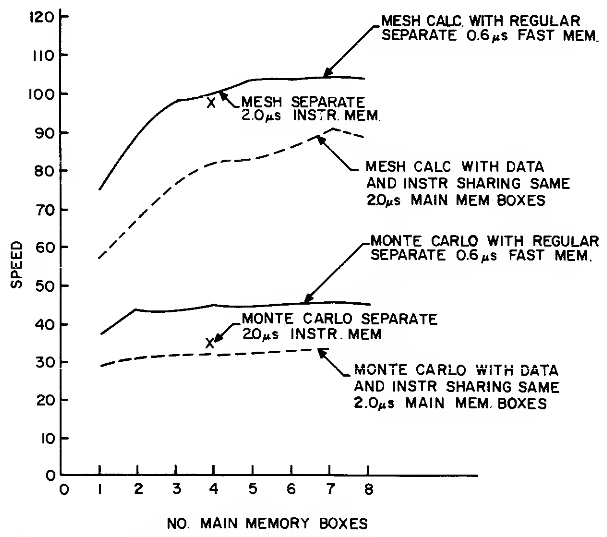


Fig. 14—Computer speed vs. number of main memory boxes: 4 level LA; 0.6 μs I AU time; 0.64 μs AU time.

The speed differential between having and not having instructions separated from data arises from delays in instruction fetches caused by the memory units being busy with data. The size of this effect varies from problem to problem, being less pronounced for problems which are arithmetic limited and more for logical problems.

The *X*'s on the graph show the effect of replacing the 0.6 μsec instruction memories by a pair of 2.0 μsec memories. The resulting performance change is small for the Mesh problem, which is arithmetic limited, but large for the instruction-fetch limited Monte Carlo problem.

Speed vs. Arithmetic Unit and Indexing Arithmetic Unit Times

Although everyone realizes the importance of arithmetic speed on overall computer performance, it was not until the simulator results became available that the true importance of the indexing arithmetic speeds was recognized. Figs. 15 and 16 show a two-parameter family of curves giving the computer speed as a function of the *AU* and *IAU* times.

Fig. 16, in which the arithmetic time is the abscissa, shows an interesting "saturation" effect where the computer performance is independent of *AU* speed below some critical value. Thus it makes no sense to strain *AU* speeds if the *IAU* is not improved to match. The curves in Fig. 15 show the same effect, *i.e.*, the *IAU* speed serves as a "ceiling" on performance beyond which the *AU* speed cannot pass.

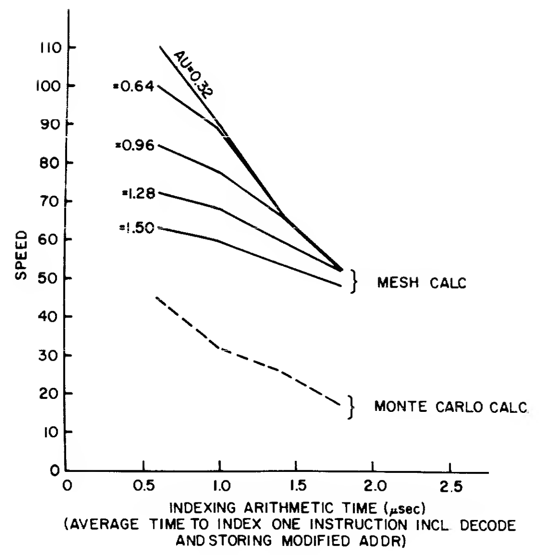


Fig. 15— Computer speed vs. indexing arith. times for various arithmetic unit times: 4 main mems. 2.0 μs; 2 fast mems. 0.6 μs; 4 levels of look-ahead.

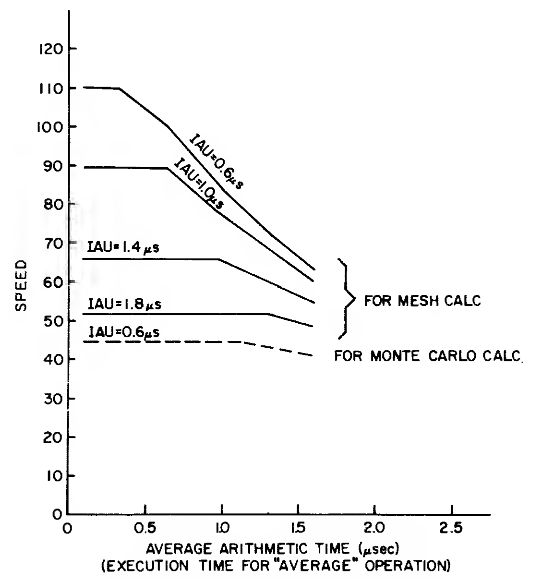


Fig. 16—Computer speed vs. arithmetic times for various indexing arithmetic unit times: 4 main mems. 2.0 μs; 2 fast mems. 0.6 μs; 4 levels of look-ahead.

Arithmetic Unit Efficiency

One fallacy which is frequently quoted is that the goal of improved computer organization is to increase the arithmetic unit efficiency. Actually there are two reasons why this is not the goal in itself. The first is that arithmetic efficiency depends strongly on the mixture of arithmetic and logic in a given problem so that a general purpose computer cannot hope to give equally high percentage utility to all. The second reason is that the simplest way to increase the arithmetic unit efficiency in any asynchronous case is to slow down the arithmetic unit.

The real goal in improved organization is maximum overall computer performance for minimum cost. One will tend to increase the arithmetic unit speed as long as its percent efficiency is reasonable for a variety of problems. One will stop this process when the overall performance gain no longer matches the increase in hardware and complexity. Thus the arithmetic unit efficiency is a by-product of this design process, not the prime variable.

Speed vs. Concurrent Input-Output Activity

Because of the relative time scales of *I/O* activity and the *CPU* processing speeds, the simulator cannot take account the availability or non-availability of data from *I/O* on the program being run. However, we can observe the effect on the computation of the *I/O* devices operating at different rates simultaneously with computing.

Using the STRETCH control word philosophy, it is possible to have a number of input-output units operating at the same time the Central Processing Unit is running. The Basic Exchange can reach a peak rate of 1 word every 10 microseconds. The high speed disk normally operates at 1 word every 4 microseconds. Since the mechanical devices take priority over the *CPU* in addressing memory, the computation slows down because of memory-busy conflicts.

Fig. 17 shows an example of how internal computing speed is slowed as the *I/O* word rates are varied continuously. At the theoretical "choke off," the *I/O* devices take all the memory cycles available and stop the calculation. Notice that this condition can never arise for any *I/O* rates presently attainable.

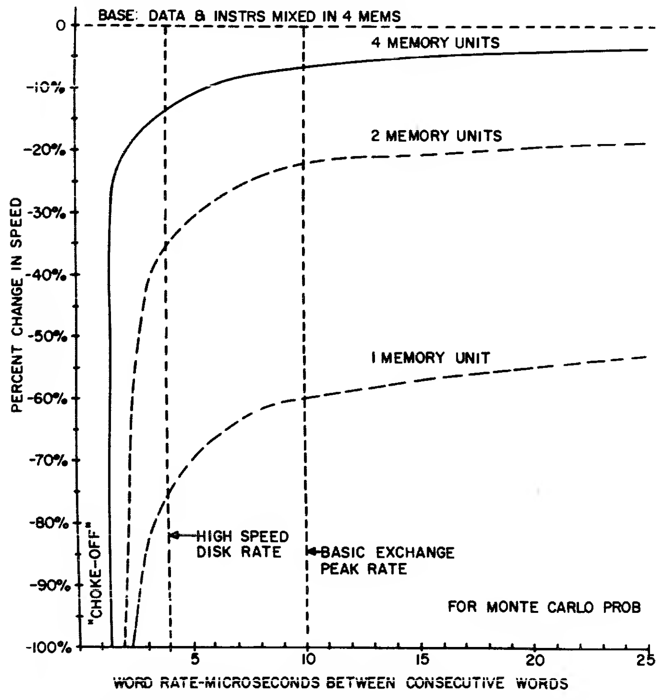


Fig. 17—Internal computing speed. Percentage reduction in speed caused by input-output devices referencing memory at different rates while the calculation is proceeding.

A STRETCH system with only 1 or 2 memory units has less performance than a larger one for three reasons: (1) The top speed of the system is reduced by the loss of memory overlap, (2) it has a larger *I/O* penalty when *I/O* is run concurrently with the computation, and (3) the smaller amount of data which can be held in the memory at one time increases the amount of *I/O* activity needed to do the job. Note, however, that increasing the memory size on a computer of conventional organization only improves the third area.

A Study of Branching on Arithmetic Results in Stretch

One penalty of the non-sequential preparation and execution of instructions used in STRETCH is that if there is a branch in the problem code it spoils the smooth flow of instructions to the indexing arithmetic unit. Any branch in a program will cause some delay, but the most serious ones are the branches on arithmetic results which cannot be detected by the indexing arithmetic unit in advance.

There are two fundamental ways in which branches on arithmetic unit results can be handled by the computer.

(1) The computer can stop the flow of instructions until the arithmetic unit has completed the preceding operation so that the result is known, then fetch the next correct instruction. This places a delay on every *AU* result branch whether taken or not.

(2) The computer can "guess" which way the branch is going to go before it is taken and proceed with fetching and preparing the instructions along one path with the understanding that if the guess was wrong, these instructions must be discarded and the correct path taken instead.

A detailed series of simulator runs were made to study this situation and to decide which way STRETCH should be designed. Some of the general observations were:

(1) The performance variation in a problem with considerable arithmetic data branching can vary by approximately ± 15% depending on the way in which the branches are handled.

(2) Holding-up on every branch seems to be less desirable than any of the guessing procedures. Some time is lost whenever a branch is executed rather than proceeding to the next instruction. Unless there is an unusual situation which there is a very large probability that the branch will always be taken, the least time will be lost if one assumes that the branch is *not* taken.

(3) The theoretically highest performance would be obtained if each branch had an extra "guess bit" which would permit the programmer to specify which way he estimates each branch will most likely go. However this would place a considerable extra burden on the programmer for the gains promised. (It also uses up many valuable OP codes.)

(4) It is realized that there is a "feedback" in such decisions because the way in which the machine guesses the branches will influence future programmers to write their codes to take advantage of the speed gain. The result is that the statistics of the future will be biased in favor of the system chosen for the machine, and thus "prove" that it was the right decision.

Acknowledgment

The general idea of "look-ahead" was under consideration by many people in IBM before the authors became involved. What is represented here is a realization of the detailed logic of look-ahead, similar enough to STRETCH for practical simulation purposes. The actual precise detail of the logic as it appears in the STRETCH computer represents, of course, the accomplishments of many individuals in the STRETCH project.

Discussion

M. Rubinoff: What happens to the look-ahead process if a sequence of branch instructions is programmed, such as in the binary selection of one of many subroutines? An example is the selection of the desired piece of a piece-wise function approximation.

Dr. Kolsky: If it is an unconditional branch then it takes a correct path.

Mr. Rubinoff: These are conditional?

Dr. Kolsky: The machine makes the assumption that the branch is *not* taken. If the path is not taken then the branch time is covered up.

M. S. Maxwell (*US Naval Weapons Lab.*): Discuss maintenance on diagnostic programs to insure proper operation of virtual memory.

Dr. Kolsky: The STRETCH machine has as one of its unusual features a part of the interrupt system capable of recording the status of the machine at the instant the interrupt occurs, so that one gets a "snapshot" of the machine as of that moment. This occurs so you do not have to go back and duplicate the error by running the program over and over again. I think you can see by the way the virtual memory operates that it would be very difficult to duplicate the error again. This feature, whereby a snapshot is made at the time of the error occurs, enables the engineers to go over the records and determine exactly what it was that caused the failure. Of course, the machine has a very elaborate checking mechanism as was described by Erich Bloch in his paper yesterday.

J. Anderson (*Burroughs*): Is the addressing of STRETCH's main memory sequential within a memory unit or sequential across several memory units?

Dr. Kolsky: The Los Alamos machine has six memories. Two are alternating and the other four are sequential across all four.

R. MacIntyre (*Bausch & Lomb*): Is the virtual memory addressable in case of a branch?

Dr. Kolsky: No, it is completely unavailable to the programmer. You can see that one would get into some rather tricky logical problems if it could be addressed. We discussed this at length and one gets into a terrible spider web of logical complications when one does that.